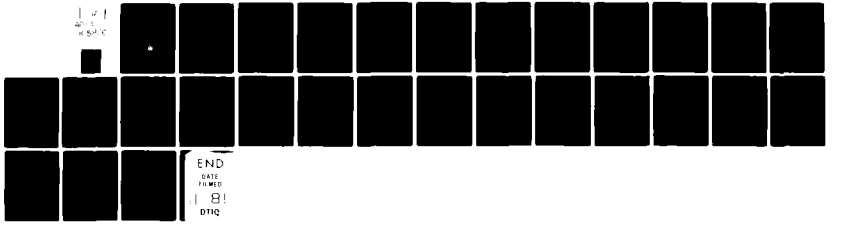
AD-A105 826 MARYLAND UNIV COLLEGE PARK DEPT OF INFORMATION SYSTE--ETC F/G 9/2
ON THE SPECIFICATION OF DATABASE SEMANTIC INTEGRITY.(U)
SEP 79 M L BRODIE DAAG29-78-G-0162
UNCLASSIFIED NL



ON THE SPECIFICATION OF DATABASE SEMANTIC INTEGRITY*

SEP 1981

THE DEPARTMENT OF
INFORMATION SYSTEMS MANAGEMENT

UNIVERSITY OF MARYLAND
COLLEGE PARK, MD.
20742

Table of Contents

Letter on file

A

Abstract

Semantic integrity is fundamental to the correct application and use of database systems. A database exhibits semantic integrity if it is logically consistent and complete with respect to the "real world" application being modelled. Although the evaluation of semantic integrity relies on intuition to a large degree, database models should facilitate its demonstration. To meet these requirements database models must be rich enough to permit the specification of the necessary semantics and to support the verification and validation of consistency.

Database, programming language, and artificial intelligence concepts are integrated and extended to provide tools and techniques for improved database semantic integrity. Artificial intelligence concepts are applied to improve the semantic power of database models. Data type concepts are extended to accommodate databases and vice versa. The result is a semantically rich database model, based on data type concepts, and a schema specification language which integrates these concepts. This approach permits data type concepts to be applied directly to databases. It is argued that database semantic integrity can be improved through specification and verification tools and techniques based on data type concepts. The role of axiomatization to formalize both the database model and its data language is described. Software engineering tools are applied to database design.

Keywords and Phrases: database, data type, database semantics, database schema, database model, programming languages, abstract data types

CR Categories: 4.33, 4.34, 4.22, 5.24

On the Specification of Database Semantic Integrity

Michael L. Brodie

Department of Computer Science

University of Maryland

College Park, MD 20795

1. Introduction

Database concepts have evolved in a pragmatic, informal way somewhat independent of related developments in programming languages, artificial intelligence (AI), and other areas. This may be due, in part, to the practical aspects of data processing and to the genuine need for ideas and approaches not offered in other areas. The result has been a myriad of informally defined database terms and a multitude of database models with little theoretical base. However, the study of databases is an application area which shares many problems and goals with other computer science disciplines. In particular, problems of information representation and reliability are pervasive. The database area has reached a mature stage and it may now be appropriate to apply some well-known results from related areas to databases in order to formalize some basic database concepts and to address problems of database semantic integrity.

Semantic integrity is fundamental to the correct application and use of database systems. A database application exhibits semantic integrity if the properties represented by the database together with its schema and transactions is consistent and complete with respect to the "real world" application being modelled. This intuitive notion can be made more precise by means of a written specification of those properties that must be represented. A specification acts as a definition of the semantics or meaning of the application and provides a basis for verification.

Database models can be designed to support the specification and verification of database semantic integrity. Such database models must permit the specification of the desired properties and facilitate the verification and validation of those properties. A schema specification is verified by ensuring that the rules of the database model are obeyed and that no descriptions are incomplete or are inconsistent. A database is validated by ensuring that the data values satisfy

the specified properties. A database application that has been verified and validated with respect to a specification is said to exhibit semantic integrity.

Unfortunately, database tools and techniques currently available to ensure semantic integrity are inadequate. Conventional database models are semantically weak; they cannot be used to express many necessary properties using non-procedural means [Schmid and Swenson 1975; Brodie and Scnmidt 1978; Kent 1979]. A large number of semantic database models have been introduced [Abrial 1974; Roussopoulos 1975; Biller and Neuhold 1978; Hammer and McLeod 1978] to address some of these issues, however, there has been little attempt to give formal definitions of the semantics of these models.

The consequences of these problems place severe restrictions on the design, construction, analysis, and use of databases. Since the semantics of database models are not rich and are poorly understood, it is difficult to design databases with the desired properties. Means beyond the database model are required to express and enforce various properties. Typically, these methods are procedural and are applied in an ad hoc manner. Databases with such constraints are difficult to construct and costly to run. Due to the lack of a theoretical base, analysis for consistency and completeness, and comparisons with other schemas are very nearly impossible. All of these factors contribute to the difficulty for a user to comprehend a database adequately.

This paper contributes to solutions for some of the above problems. These contributions draw on results from programming languages and AI both of which have problems and goals in common with databases. Databases and programming languages are concerned with data reliability -- ensuring that the data in a system obeys specified properties [Hoare 1975]. From programming languages we apply data type concepts to databases in order to formalize some database concepts, to improve the semantic power of data models, and to integrate some aspects of databases and programming languages. Databases and AI are concerned with the representation of knowledge [Wong and Mylopoulos 1977]. From AI we emphasize the importance of intensional information in the schema. A conceptual framework is given in section 2. Section 3 introduces abstraction mechanisms for programming languages in order to accommodate the inherently complex and semantically rich database problems. A data type specification model is defined. Section 4 presents a new, semantically rich database model based on an algebra of data types. Finally in section 5, Beta, a database schema specification language, is discussed.

2. Conceptual Framework and Approach

2.1 Conceptual Framework

Figure 1 illustrates four progressively more abstract levels in which to consider database description and storage.

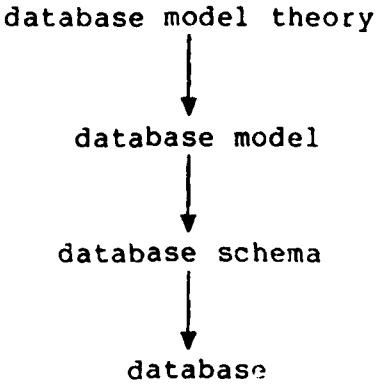


Figure 1: Conceptual Framework for Database Description and Storage

A database is a collection of data values that satisfy a given schema; it represents a state of some application at a fixed time. A database schema is a collection of rules that describe and restrict a class of databases; it represents all possible states of the application. A schema must obey the rules of a particular database model. A database model includes a collection of rules (expressed in formally definable data structures) that define a class of schemas. Database model theory is the level at which all database models may be discussed and compared; several such theories are now beginning to emerge.

2.2 Structure versus Behaviour

Typically, the levels of the conceptual framework have been considered in terms of structural, rather than behavioural, properties. Structural properties of a database determine the possible databases or database states. For example, a schema describes a class of databases by means of declarative, non-executable data definition language (DDL) statements. Behavioural properties of a database determine the possible state transitions which are described procedurally by executable data manipulation language (DML) statements. Generally, behaviour is not defined in the schema. In many areas of computer science it has been found that some properties can be described either as structure or as behaviour. This duality can be seen in the lack of distinction between DDL and DML in both the relational and DBTG data models.

Structural descriptions cannot capture all the properties of a database application, however, for several reasons it is important to investigate the extent to which semantics can be expressed and maintained via structure. First, the emphasis in the database area has been on data with persistent structural properties rather than on operations which may change through time. Second, some structural descriptions are more simple, concise, and abstract than are their behavioural counterparts. For example, a structural relationship between two entity types might affect all modification operations on each entity type. Third, structural descriptions are more conveniently understood and analyzed than are behavioural descriptions. Finally, well-known tools exist for the description and analysis of structural properties; these can be used for the specification, verification, and maintenance of database semantics.

2.3 Database Constraints

The most common method of increasing the semantic power of a database model is via constraints. Constraints have been viewed as those properties of the "real world" application that the database must obey but which cannot be expressed directly in terms of the database model being used. Since the ability to express various properties varies from one database model to another, the constraint concept has been database model dependant. This view is troublesome for the design, analysis, and comparison of schemas, particularly for multiple, coexisting schemas [ANSI/SPARC 1977]. Here, we define a constraint to be any property of the "real world" application that must be represented in the database for logical completeness. This definition brings several insights to the analysis of databases. Some constraints are inherent in the database model being used; some are stated explicitly via DDL or DML statements; still others are implicit, inferred from inherent and explicit constraints. For example, the dependency constraint inherent in the hierarchical database model must be stated explicitly when using the relational model, and the closure of a set of functional or multivalued dependencies [Bernstein 1976] is an implicit constraint which can be inferred from a smaller set of explicitly stated dependencies.

2.4 The Data Type Approach

In their development from data files, databases have not used the data type concept explicitly, whereas data types have been fundamental to the development of programming languages. The advantages of abstract data types [Liskov and Zilles 1974,75; Guttag 1975] have been considered, briefly, for databases [Hammer 1976; Brodie and Schmidt 1978]. Also, Schmidt [1977] has considered data types for the definition of relations in an extension to Pascal.

Data types provide many benefits for dealing with the structural aspects of databases. This is due to the close correspondence between the main purposes of data types and the needs of databases, namely:
(1) formal description of structural (and some behavioural) constraints,
(2) automatic maintenance of constraints, and
(3) recognition and generation of instances of abstract objects.

In the sequel, it will be shown that many data model criteria [McGee 1976] can be addressed via data type concepts. Traditionally, data types have been the meeting point for both logical (user) and physical (implementor) requirements. Although there are many advantages for database implementation, we will consider the advantages for database design and analysis necessary for ensuring semantic integrity.

3. Data Types in Programming Languages

3.1 Abstraction

Abstraction, the most powerful intellectual tool, has been used extensively in computer science in developing our understanding of complex phenomena [Hoare 1972]. In this section, abstraction is used to extend data type concepts and to relate them to the conceptual framework for databases. The ideas are then used to develop a data type specification model.

We will be concerned principally with two forms of abstraction: generalization and aggregation [Smith and Smith 1977] which are based on the is-a and part-of relationships in semantic networks [Roussopoulos 1975]. Generalization enables a class of objects to be thought of as a single generic object. In particular, types can be generalized from tokens. A token is a data value which can be contained in an instance of an abstract object whereas a type is an abstraction that stands for a class of tokens. Aggregation enables a relationship between (constituent) objects to be considered as a higher level aggregate object. For example, the generic type furniture_item stands for a class of tokens including actual chairs and tables while the aggregate chair may have the constituents legs, seat, and back.

Generalization aids our understanding of phenomena by allowing classification. Objects are classified so as to emphasize their similarities and to ignore their differences. Generalization can be applied repeatedly to types resulting in a generalization hierarchy which has a downward inheritence property. Each property of a generic type is inherited by all its subtypes, however, a subtype may have properties that distinguish it from other subtypes. In the furniture example, chairs and tables have similarities as furniture and differences which distinguish chairs from tables.

Aggregation aids our understanding of a phenomenon through structure. The structure of an object can be seen in terms of the relationship amongst its constituents. The repeated application of aggregation results in an aggregation hierarchy which has an upward inheritence property. Each property of a constituent becomes a constituent property of the aggregate. The inverse process, stepwise refinement, produces an hierarchical breakdown of an object into its constituents which in turn can be broken down further.

Generalization and aggregation can be used in a complementary fashion to express both the structure and classification of objects: aggregates can be classified and generic objects can be structured. Hence, a complex phenomenon can be considered in layers of abstraction. The major advantage of abstraction is that properties can be considered in isolation and inessential details can be ignored.

3.2 Abstraction and Data Types

In programming languages, aggregation has long been used to express the structure of data types, however, typically generalization has not been used to express relationships amongst types. Aggregation is a fundamental composition rule used to define structured types, such as records and arrays, from simpler constituent types. We propose to extend the data type concept by including generalization as a powerful composition rule for data types. The resulting data types are both semantically rich and facilitate the representation of complex objects frequently found in database applications.

Data type concepts can be described in terms of three forms of generalization. Token generalization is applied to tokens and produces data types. Type generalization can be applied to data types resulting in more complex types. Category generalization is applied to data types to produce data type categories which are sometimes called data structures. For example, the record data type category is a generalization of all record types, e.g., employee and department. In turn, the employee record type is a generalization of data values, e.g., employees Fred, Tony, and Yannis.

Observing that each level of the conceptual framework is related to the next by generalization, we can relate data type and database concepts as shown in Figure 2, A database is a collection of tokens in instances of data types in a schema. A schema is a collection of data types which represent the constraints of the "real world" application. A database model includes a formally definable collection of data type categories. Database model theory corresponds to the theory of data structures.

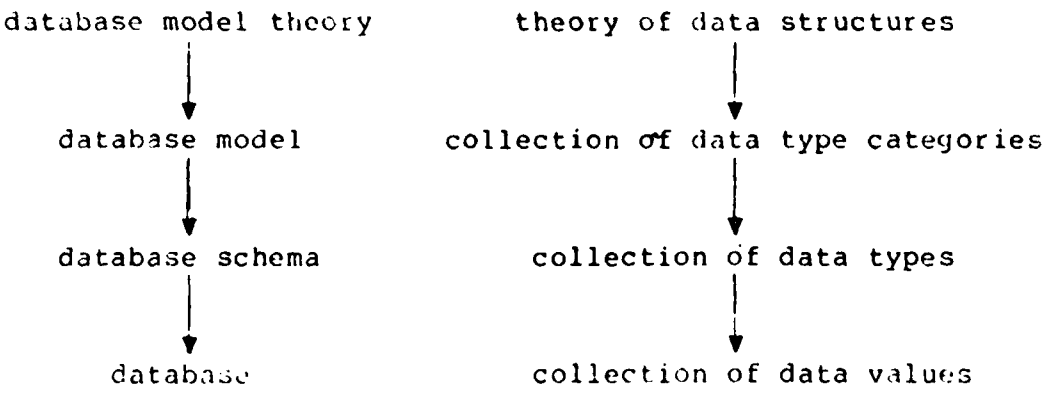


Figure 2: Correspondence between Databases and Data Types

3.3 A Data Type Specification Model

To conclude this section, abstraction is used to define the data type concept and a data type specification model.

Hoare's stages of the abstraction process [Hoare 1972] can be used to define the essential aspects of the data type concept. The aspects are: (1) Abstraction: the object, the type itself, that results from abstraction applied to classes of types and tokens. (2) Representation: the set of symbols chosen to stand for the type and its tokens. There is both a storage representation, the mechanisms used to map tokens into storage, and a logical representation, the mathematical properties of the symbols used. (3) Manipulations: the transformation rules for representation, the specific operators for tokens of the data type. (4) Axiomatization: the rigorous statements of the definitive properties of all instances. For a given type, these aspects consitute a data type specification.

We now present a model for specifying data types, based on the above development and on a model proposed in [Gotlieb and Gotlieb 1978]. The model presented here has been used to define [Brodie 1978], formally, the database model presented in the next section.

A data type name, say t, is used to denote the abstraction. The logical representation is defined by the triple: <V(t), ID(t), C(t)> where V(t) is the value set (i.e., a set of tokens) for t, ID(t) is the identification rule for variables of t, and C(t) is the set of constituent types used to compose t. The storage structure is defined by the triple: <ID(t), M(t), SMF(t)> where M(t) is the set of memory locations in which tokens of t may be stored and SMF(t) is the storage mapping function that maps identifiers from ID(t) to memory locations in M(t). The SMF(t) is a principal part of the instance-of or binding relationship between t and its instances which contain tokens. The manipulations O(t) is the set of operators for which values from V(t) may serve as operands. O(t) corresponds to the syntactic specification in [Guttag 1975]. The axiomatization (c.f. [Hoare 1972] and [Guttag 1975]) is defined by the triple: <AC(t), AV(t), AO(t)> where AC(t) is a set of axioms defining the way in which t is composed from its constituent types C(t). AV(t) is a set of axioms defining the properties of values in V(t). AO(t) is a set of axioms that define the operators named in O(t). AO(t) corresponds to the axioms proposed in [Guttag 1975] which give the semantics of operations, in some sense. In summary, a data type, t, can be specified by the tuple: <V(t), C(t), ID(t), SMF(t), M(t), O(t), AC(t), AV(t), AO(t)>.

4. Data Types in Databases

4.1 On Differences Between Data Types and Databases

There are five major differences between data types and databases. The differences concern data independence, problems of scale, type checking, dynamic aspects of instances, and data sharing. These differences can be resolved so that data types can be applied to databases.

A fundamental difference between programming languages and databases has been the relative importance of algorithms and data. Stepwise refinement advocates postponing data structure decisions until algorithms are designed. In databases, data structures must be designed without knowing what algorithms will be applied. The approach to data types and databases is now changing. One of the principal goals of both databases and abstract data types is an ability to design and alter representations without unduly impacting programs or data. This necessitates a flexible binding mechanism.

Since database systems typically consist of several large programs accessing large volumes of highly interrelated data, problems of complexity due to scale and semantics are inherent. Due to the need for data independence and data relatability, scope and modularity, the two data type concepts most effective for managing large numbers of data types, do not readily apply to databases. The multiple schema concept is still unclear and problematic [Pelagatti et al. 1978]. Aggregation and generalization have been applied expressly to to address problems of semantics and scale. They facilitate the construction of abstract views which exclude inessential details. Abstraction mechanisms also permit the expression of semantic not expressible using other database models.

A third difference is that the weak type checking and compatibility rules of most programming languages are not adequate for database problems. For example, in Pascal [Jensen and Wirth 1974] and Euclid [Lampson et al. 1977] the following two types are the same, hence, compatible:

```
type  height = 1..100;
type  weight = 1..100;
```

Consequently, meaningless relational algebra operations cannot be detected automatically. Hence, strict type compatibility rules are required.

A fourth difference concerns the dynamic aspects of instances. Typically in programming languages, there is a one to one correspondence between identifiers and instances; their number is known in advance. A database is initially empty; instances are created and possibly destroyed in a series of program invocations. Hence, a dynamic indentification mechanism for instances is required.

A final difference is that the persistence and sharing of data is fundamental to databases but is seldom supported in programming languages. Typically in programming languages when the scope of a variable is exited, the instance is lost. Imported, exported, and own variables have been introduced (e.g., see Euclid [Lampson et al. 1977]) to reduce the sharing of instances amongst scopes. In databases, successive or concurrent invocations of the same or different programs must be able to access the same data values.

The differences concerning a flexible binding, dynamic identification, sharing, and persistence can be resolved by including the database concept of key as an identification rule for data type instances. A key is some subset of the constituent types of a data type whose values uniquely identify instances. One consequeence of this inclusion is that prime constituents, those taking part in a key, may not be altered without destroying the containing aggregate instance and creating a new one. Another typical consequence is that keys cannot assume null values since this would violate the uniqueness rule. The inability to assume given values or to have certain components altered is, generally, not the case for instances of data types.

4.2 The Limited Generic Database Model

This section presents a database model which incorporates the above solutions. The concept of a data type algebra is introduced and is used to define the semantics of the database model. The model is based on two simple concepts, objects (or entities) and relationships used to compose or relate objects. The database model incorporates the essential rather than all properties of the hierarchic, network, and relational database models, hence, it is called the limited generic database model (LGDM).

For a given database model, a data type algebra consists of the data types that correspond to the data type categories of the model plus the type operators used to compose those types. A data type algebra is used to specify database schemas. A definition of a data type algebra is also a definition of the properties of all schemas within the datbase model. These properties have been called the semantics of the database model. The LGDM is described here in terms of the LGDM data type algebra. First the data type categories are presented, then the type operators are given. A more detailed definition of the LGDM and of the data type algebra is given in [Brodie 1978].

The LGDM has two categories of unstructured types, base types and interpreted types, and four categories of structured types: attributes, object types, map types, and set types.

The base types: integer, Boolean, char, enumeration, and subrange are taken from Pascal. Their properties are defined by well known axioms [Hoare 1972]. The base type string is a

sequence of characters much like Pascal's packed array [1..n] of char.

Interpreted types are introduced to permit the distinction between semantically different types based on the same underlying type. An interpreted type is a particular use of a base type that is distinguished from all other types having the same underlying type by means of a type identifier. The identifier may be considered a units name. Two types are compatible if and only if they are based on the same interpreted type. The value set, the value axioms, and the relational operators are inherited from the base type, although the order axioms and the corresponding relational operators can be excluded by specifying "unordered". The remaining type properties are not inherited. For example, the interpreted type (denoted by @)

```
type employee_number = @(1..100) unordered;
```

does not inherit arithmetic or relational (except <> and =) operators which in this case have no meaning. The interpreted type concept is based on opaque and transparent types in [Morris 1973], on labelled modes in EL1 [Wegbreit 1974], and on the interpreted type in [Schmidt 1978].

An attribute is a particular use, within an object type, of an interpreted type or another object type and an identifier. An attribute identifier is used to denote the role it plays within the object type. An attribute is simple if the underlying type has one constituent, otherwise it is non-simple. Non-simple types provide a degree of abstraction and modularity; they can be treated as aggregates in several object types.

An object type is an aggregate of attributes. Its value set is a Cartesian product of the value sets of the constituent attributes. An object type must have at least one key so that instances can be identified uniquely. Instances of object types can have independent existences within a database, whereas, attribute instances exist within the context of an object type instance. No object type instance contains a variable number of tokens, hence, they follow the principle of first normal form which has advantages for both data types [Hoare 1972] and databases [Codd 1970]. An object type instance corresponds to a tuple in the relational database model.

A map type represents a binary relation between two, possibly identical, object types. The map type is introduced for both semantic and performance improvements over the join operation in the relational algebra. Each binary relationship between object types must be defined explicitly in a map type thereby reducing the possibility of creating meaningless associations. Maps represent relationships as distinct from objects. They indicate the needed access paths. Map types can be used to represent single member DBTG owner-coupled sets, joins, and hierarchic relationships.

A set type is a powerset of its constituent object or map type. At a given time, a set type is restricted to having only one instance which is a set of instances of the underlying type. An instance of a set type based on an object type corresponds to a relation in a relational database.

A data type algebra has two kinds of operators: schema level operators, called type operators, and database level operators, called token operators. Type operators are used to define new types from existing types. For a type T, a set of type operators forms the composition axioms or rules, AC(T), used to compose T from its constituent types, C(T). Token operators produce new tokens from tokens in instances in a database. For a type T, a set of token operators, with syntax given by O(T) and behavioural semantics given by AO(T), define the legal manipulations of values of V(T). Type and token operators are related intimately. Type operators can be defined by applying a sequence of token operators to elements of the value sets of the constituent types thereby producing a value set for the new data type. For a complete specification of a data type, both token and type operators must be defined.

In keeping with our structural approach, we consider only the following token operators which are necessary to define the LGDM type operators, i.e., those for value selection and testing. The selection operator [Hoare 1972] is used to select a component value from an instance of a structured type. The relational (i.e., <,<=,=,>=,>,<>) and the Boolean operators are defined in the conventional way over unstructured and Boolean values, respectively. The de-interpretation operator takes a value of an interpreted type and produces the corresponding value of the underlying base type. It is similar to the LOWER primitive function in EL1 [Wegbreit 1974] but can be used only to alter the compatibility of the value of a variable when compared with a constant value. The above operations can be combined to form predicates which are logical operators. Finally, the token restriction operator is a logical operator which produces the argument value when the predicate is satisfied, otherwise it produces the null value. The remaining token operators (e.g., insert, delete, update, and arithmetic operators) are not considered here.

The LGDM type operators form a set of composition rules which can be used to express the complex structural relationships required to represent the semantics of databases. They are based on aggregation and generalization, and on the relational algebra applied at the schema level rather than at the database level. The type operators and the data types of the LGDM form the data type algebra which generalizes the concepts of data type and data structure discipline, e.g., as found in Pascal. The eleven type operators are now introduced.

The interpretation operator takes one base type as an argument and produces an interpreted type. It is similar to the mode operator in EL1 [Wegbreit 1974].

The Cartesian product operator produces an object type from two or more attributes. Instances of the resulting types are tuples unlike the relations (i.e., sets of tuples) that result from the extended Cartesian product in the relational algebra. The result of the Cartesian product operator is similar to the Cartesian product data type in [Hoare 1972].

The union, intersection, and difference operators each produce an object type with a value set which is, respectively, the union, the intersection, and the set difference of the value sets of the two argument object types. Two object types are compatible under these operators if they have the same key, i.e., compatible prime attributes.

The projection operator forms a new object type from a subset of the attributes of another object type. The resulting value set is determined by applying the selection operator to each element of the argument value set to select the component values of the projected attributes with duplicate values deleted. This corresponds to the relational projection.

The type restriction operator produces an object type that is restricted to those elements of the argument value set that satisfy a logical expression. The resultant value set is determined by applying the token restriction oprator to each element of the argument value set. The type restriction operator coresponds to the relational restriction.

The division operator produces a data type with a value set which is a subset of the value set of the first of the two argument object types. The resulting value set is computed by applying the relational division to the two arguments based on some specified, compatible attributes.

The map operator relates two object types via a binary relation and results in a map type. The resulting value set is determined by applying the relational join operation to the two argument value sets based on some specified, compatible attributes.

The powerset operator produces a set type with a value set which is the powerset of the value set of the argument map or object type. An instance of the resulting set type is a set of values of the argument type.

Finally, the assertion (type operator) is the most general. An assertion is an applied predicate calculus expression used to express a logical relation over one or more object types. We can regard this operator as having the effect that, a new data type is produced with a value set that is those combinations of the elements of the value sets of the argument object types that satisfy the assertion. Strictly speaking, no such single type results. A major advantage of assertions is that complex relations and views can be defined declaratively over existing types without affecting their basic structure or composition -- a form of data independence.

Type operators are used to express aggregation, via the Cartesian product, and various forms of generalization. Generalization is a concise method of expressing various properties. If type A is a generalization of type B, then every instance of B is also an instance of A. B is said to be dependent on A, and the properties of A are inherited by B. Generalization can be expressed by the set opreators: union, intersection, difference, and division and by restriction and projection (as long as one key is projected).

A schema that results from relationships expressed using the data type algebra is a network of data types. The network consists of one or more generalization hierarchies possibly related via binary relationships between types in the hierarchies. Each type can in turn have an aggregation hierarchy of constituent types. Although the resulting schema is complex, reflecting the semantics of the database, its construction and understanding are greatly simplified by abstraction which enables small aspects to be considered in isolation.

5. Beta: a Schema Specification Language

5.1 The Purpose and Basis of Beta

This section presents the underlying concepts and purposes of Beta, a language for the specification of consistent schemas based on the LGDM. The purpose of a schema specification is to give a precise but succinct description of the constraints that constitute a schema. Hence, Beta can be used to design and express the semantic or logical aspects of a schema while ignoring semantically irrelevant details such as representation. A schema and its specification are consistent if all the constraints can be proven to be mutually satisfiable. The formal definition of Beta can be used as a basis for consistency verification.

Beta provides a programming language framework in which both database and data type concepts are integrated. The three underlying purposes of data types -- constraint definition, automatic constraint maintenance, and instance recognition -- are used to meet the needs of databases. The simple data types of the LGDM are used to specify the complex constraints of database appliciations. Beta draws heavily on both programming language and database concepts. It is based on the type concepts, syntax, and axiomatization of Pascal [Jensen and Wirth 1974]. Beta contains some clarifications of (e.g., same type) and extensions to (e.g., legality assertions) Pascal as defined in Euclid [Lampson et al. 1977] for the purpose of verification. Beta extends the ability of Pascal/R [Schmidt 1977] to express constraints via structural means. Finally, Beta is based on the LGDM, hence, it contains the data type algebra.

The design of Beta was guided by several interrelated goals. In particular, the data reliability and structuring goals of [Hoare 1975] and those for type extensions and modes of [Parnas et al. 1976] were considered. The primary goal, however, was to address the data model and database problems described in sections 1 and 2. Hence, Beta addresses three main goals:
1) to provide a semantically rich schema specification facility,
2) to provide a formal definition of data base semantics as a basis for schema analysis, and
3) to provide a basis of a database design methodology.

This section describes four aspects of Beta. First, the semantic power of Beta is illustrated by examples. Second, the formal aspects of Beta are discussed. Third, some programming language concepts are presented. Finally the use and benefits of Beta are discussed for database design.

5.2 The Semantics of Beta

This section demonstrates the semantic power of Beta by illustrating some of the expressible constraints with example specifications. The examples are drawn from a university database involving students, tutors, professors, and courses.

Using Beta, schemas can be specified in a modular fashion. One may concentrate on simple constraints specified in simple types over which more complex constraints can be specified. The elementary specifications consist of constants, base types, and interpreted types. These are used to construct the more complex attributes, object types, and map types over which assertions, the most general specifications, can be made.

Base types are used to specify value sets of atomic types. Value sets can be enumerated explicitly or specified as Booleans, characters, subranges, or strings. The minimum and maximum length of strings can be specified. A specification of an interpreted type, denoted by @, is used to interpret a particular use of a base type. In example 1, postal_code and course_#type are incompatible to ensure their semantic difference.

Example 1.

```
type social_insurance# = @(550000000..850000000);
     postal_code       = @string[6..6]of char;
     course_#type      = @string[6..6]of char;
     statustype        = (full_time, part_time) unordered;
     gradetype    = (incomplete, F, pass, C_minus, C, C_plus,
                     B_minus, B, B_plus, A_minus, A, A_plus);
```

The central type category, the object type, represents a class of entities, the only instances capable of independent existence. Object types are aggregates of attributes which denote the roles played by underlying types. In example 2, person is an aggregate of name, number, sex, address, and title. Attributes can be simple, e.g., sex and title, or can themselves be aggregates, e.g., addresstype may be composed of street, city, province, and postal_code. Other constraints expressible over object types are keys, e.g., name and number are equivalent keys for person, as well as functional (->) and multivalued (->->) dependencies.

Example 2.

```
type person = object name:  nametype;
                     number:  social_insurance#;
                     sex:  (male, female) unordered;
                     address:  addresstype;
                     title:  (professor, tutor, student)
                     keys name, number
                     dependencies name->address|sex|title
              end object;
```

As can be seen in example 3, generalization applied to object types expresses several constraints in a succinct, modular

manner. Roles [Bachman and Daya 1977] played by object types can be specified, e.g., person can play employee and student roles. Generalization relationships between object types are expressed by means of object constructors, e.g., a student is a particular kind of person. and set expressions, e.g., tutor is the intersection of student and employee. Through generalization inheritence, student and employee have all the properties of person. Additional properties are used to distinguish object types from their generic types, e.g., employee has deduction and salary plus an additional multivalued dependency. Generalization implicitly specifies a dependency constraint between object types, e.g., a student (role) exists only if a corresponding person exists.

Example 3.

```
type employee =
      object [each p in person where p.title = professor
                                  or p.title = tutor];
            salary,
            deduction:  moneytype
            dependencies person->->salary
      end object;

   student =
      object [each p in person where p.title = student
                                  or p.title = tutor];
           status:  statustype;
           academic_year: (first, second, third,
                                  fourth, special)
      end object;

   tutor = object employee*student end object;
```

A set type specification associates an identifier with the implicitly defined set type for the underlying object or map type. For a given database, a set type has exactly one set-valued instance, i.e., the set of existing instances of the underlying type. In a given university database, the instance of people consists of all person instances.

Example 4.

```
type people = set of person;
```

A map type specifies a binary relation between two object types by stating how an instance of one object type can be related to a quantified set of instances of a second object type. Quantification is absolute since a minimum and maximum set size is given. In example 5, each tutor tutors at least one and at most max_tutor_load courses. The relationship can be further defined by join terms in a where clause which facilitates the automatic construction of instances, otherwise, instances must be created manually using DML commands. Maps can express injective, surjective, and bijective maps. Tutors is injective; a tutor must be associated with a course but a course need not have a tutor. Since academic records depend on students and not vice versa ([0..*] means a lower limit of 0 and no upper limit)

has_academic_record is surjective. Is_enrolled is bijective, since each student must have at least one enrollment and each enrollment is dependent on a student.

Example 5.

```
type tutors = map from t in tutor
                  to [1..max_tutor_load] c in course
              end map;

     is_enrolled =
        map from s in student
            to [1..max_student_load] dependent e in enrollment
            where s.number = e.student_#
        end map;

     has_academic_record =
          map from s in student
              to [0..*] dependent a in academic_record
              where s.number = a.student_#
          end map;
```

Finally, assertions are generalized, first-order predicate calculus expressions used to specify many types of constraints. Existential (some), universal (all), and absolute (exactly n, at most n, at least n) quantification can be used to specify existence and dependency constraints over one or more object types. Relations are defined precisely by means of Boolean expressions composed of join terms (e.g., e.salary > e.deduction), restriction terms (e.g., p.salary@ > 15000), map terms (e.g., t tutors c), and scalar terms (e.g., average[...]@ < 30000).

Example 6.

```
   {professors earn between $15,000 and $30,000}
   assert
     all p in professor (p.salary@>15000 and p.salary@<30000);

    {employee's salary must exceed deductions}
   assert
     all e in employee (e.salary>e.deduction);

   {tutors must have had at least a B+ in courses they tutor}
   assert
     all t in tutor (some c in course (t tutors c and
     exactly 1 ar in academic_record (t has_academic_record ar
     and ar is_record_for c and ar.grade>B_plus))));

   {the average professor salary is less than $30,000}
   assert
     average [each p.salary for p in professor]@ < 30000;
```

5.3 Formal Aspects

Beta has been defined axiomatically [Brodie 1978] following a technique introduced by Hoare and subsequently used to axiomatize Pascal [Hoare and Wirth 1973] and Euclid [London et

al. 1978]. The main purpose of axiomatizing a programming language is to give both implementors and users a clear definition of its semantics. Data types and executable statements are defined by means of axioms, which give the underlying properties, and inference rules, which define the effect of operators. A secondary purpose of such a formalization is to provide a basis for the proofs of programs written in the language.

The main role of the axiomatization of Beta is to give a clear definition of the semantics of databases specified using Beta. This may be understood in terms of our conceptual framework. If the data type categories and the type operators of the LGDM (the data type algebra) are defined axiomatically, we can determine the possible schemas -- the properties of the legal data types. Then, we can determine the semantics of valid databases, i.e., what databases map onto given schemas. Such a formalization of Beta is necessary due to our emphasis on semantics and since several concepts, not typically available in programming languages, have been introduced to accommodate databases, e.g., keys, dependencies, and the data type algebra.

The formalization of Beta fulfills two secondary roles related to constraints. First, it defines precisely what constraints can be expressed in an LGDM schema. The axioms define the inherent and explicit constraints while the inference rules define the implicit constraints. In this way, the semantic power of the LGDM and LGDM schemas can be analyzed. Second, the axiomatization forms a basis for the verification of constraint consistency and for the validation of databases against given schemas.

The axiomatization of Beta is the rigorous statements of all properties of each data type category and of each type operator in Beta. The properties of a data type category are those properties shared by all data types of that category. Hence, the data type specification model, presented earlier, has been generalized to form a specification model for Beta. A data type category, T, can be defined formally by the axioms and inference rules required by the model, i.e., <V(T), AV(T), C(T), AC(T), O(T), AO(T), ID(T)> as defined above. V(T) and AV(T) define the possible value sets for types under T. AC(T), the type operators, define the possible compositions of T from C(T). AC(T) are defined in terms of Boolean algebra and set theory. O(T) and AO(T) define properties of token operators and ID(T) defines identification rules for types in the category T.

5.4 Programming Language Aspects

By integrating data type and database concepts in a programming language framework, Beta addresses the problems raised by both self-contained and host-language interfaces to databases. On the one hand, self-contained interfaces are not rich enough to express special purpose algorithms, e.g., list

and string processing, according to the widely accepted principles of high level languages. On the other hand, host language interfaces present users with two rather different sets of concepts, two database models, two languages, and two programming techniques. These interfaces tend to be provided through rather cryptic interfaces such as parameterized procedure calls. These issues hinder the design, construction, analysis, and use of databases. Beta demonstrates that a high level or special purpose language can be extended, through its data types, to include database facilities in keeping with the philosophy of the programming language. Hence, databases and programming languages can be related in a rather intimate way.

The data type extensions introduced in Beta require corresponding extensions to type checking techniques. Many standard type properties can be verified using conventional type checking augmented by the "same type" concept from Euclid [Popek et al. 1977]. Two types are compatible under certain operations only if they are the same, that is, if they have the same specification after all type identifiers have been replaced by their specification until only interpreted type identifiers remain. However, Beta incorporates an extension of the relational calculus, e.g., restrictions, maps, and assertions are expressed as predicates. Hence, new type checking techniques are required to verify and validate Beta schema specifications [Brodie 1978]. Due to both theoretical and run time problems (e.g., decidability and existence) practical verifiers cannot verify and validate all properties. Therefore, semantic integrity assertions (legality assertions in Euclid) should be emmitted for those properties beyond the means of the verifier. If they are verified, by some other means, and validated with respect to a given database, the schema and the database are said to exhibit semantic integrity.

5.5 Database Design Using Beta

One of the main goals of Beta is to aid database design. In terms of the above development, the problem can be expressed as follows: Given an informally described application model, the properties of entities and relationships of interest, specify them formally as constraints using Beta so that the result exhibits semantic integrity. Two of the major problems faced by database designers are those of complexity and semantics. We now consider how Beta addresses these problems and facilitates schema design.

Not unexpectedly, the principles for good schema design are precisely those for structured programming and other software engineering disciplines. A schema should be logically complete with respect to both the application and itself. It should be minimal, i.e., non-redundant, and abstract, i.e., neither users nor implementors should be restricted to particular representations. A schema must be consistent, i.e., exhibit semantic integrity. It should be comprehensible

to a user or implementor, with a minimum of difficulty. Finally, a schema should be extensible and modifiable, i.e., it should remain stable under reasonable and inevidable changes. A necessary requirement for all of these properties is a good logical structure.

The most effective tools designed to achieve these goals are: abstraction, successive decomposition, layers of abstraction, modularity, and techniques for software specification and verificatoion. These tools, developed in terms of programming languages and data types, have been built into Beta. Hence, Beta directly supports a number of software engineering techniques, not typically applied to databases, for the design of "good" schemas.

The principal design tools of Beta are abstraction, modularity, and the concepts of specification and verification. Abstraction, both aggregation and generalization, is supported directly by the data type algebra. Structural modularity is the coherent grouping of structural abstractions to form meaningful units which encapsulate and hide structural details and which can be components of other modules. Structural modularity can be achieved using the Pascal based type specifications. Together, abstraction and modularity can be used to achieve layers of abstraction through successive decomposition. The concept of schema specification aids design by introducing a step between the informal application model and the schema representation. The essential, logical aspects of a schema can be designed while the inessential details, such as representation, can be ignored. Finally, the axiomatization of Beta provides two design aids. First, the semantics of each specification can be readily determined. Second, the axioms and inference rules form a basis for verification which can be used to test the semantic integrity of intermediate and final design decisions.

A schema design methodology, based on the above tools, has been proposed for Beta [Brodie 1978]. It extends the methodology of [Smith and Smith 1977] which produces generalization and aggregation hierarchies by means of successive decomposition. Based on modelling experience in AI and network databases, it was found that schemas are typically complex networks of entity types related by aggregation and generalization, and by more general relations. Hence, one extention is to permit more complex relations expressed via maps and assertions, as well as through more powerful generalization relations (e.g., intersection, union, restiction). Rather than being hierarchies, the resulting schemas are networks. As observed in software engineering, design is not a purely top-down process; indeed Hammer [1976] argued that database design is inherently a data up process. Using successive decomposition, it is not always clear what the highest level abstractions should be, nor what modularization is appropriate. Also, redesign and inevidable change is not easily accommodated, except for further decompositions; A second extension is to permit synthesis,

the opposite of decomposition; the data type algebra supports bottom-up design, since abstractions can be composed to form higher level abstractions.

The schema design methodology is based on the abstraction processes, aggregation and generalization, and uses both decomposition and synthesis. Aggregation uses decomposition to determine the constituents of an object and synthesis to construct aggregate objects. Generalization uses synthesis to classify similar objects under a generic object and decomposition to determine distinct subsets of a generic object.

Beta facilitates database design by means of built-in software engineering tools. Problems of complexity are addressed by the simplicity of the database model (a small number of type categories and operators) and by the direct support of abstraction and modularity. Semantic problems are addressed by presenting a semantically rich, axiomatically defined database model. Both issues are addressed by verification which can be used to ensure the semantic integrity of complex schemas and to test such properties as minimality and completeness. Based on the above and other software engineering concepts, [Brodie 1979] presents a framework for the design and development of database applications.

6. Summary

Due to the pragmatic, informal and somewhat independent development of database concepts, many obstacles are faced by database designers, users, and implementors. In particular, semantic issues concerning non-representational, or abstract, structure have not been adequately addressed. Hence, there are few effective database tools for these problems. However, well-defined, widely accepted concepts and tools have been developed to address similar issues in programming languages and AI. By applying data type concepts to databases, some basic database concepts have been formalized and databases have been related to the more widely understood area of programming languages. This approach has addressed long standing discrepencies between programming languages and databases [Atkinson 1978].

Data type concepts were extended to meet the semantic modelling needs of databases. Abstraction mechanisms were introduced and structuring rules were generalized to form the data type algebra. A semantically rich database model, the LGDM, was designed to meet well-known database model criteria. Although semantic problems and their solutions are unlimited, techniques were presented to handle some semantic issues. Data type and database concepts were integrated in a programming language framework. Beta, a schema specification language, can be used to specify precisely application semantics expressible via structure. Beta and the LGDM have been axiomatized in [Brodie 1978] to give a formal defintion of their semantics and to act as a basis for semantic integrity verification. Finally, a database design methodology, based on software engineering tools supported directly by Beta, was discussed.

Although behavioural semantics were assiduously avoided in this paper and by many researchers in database semantics and schemas, behavioural properties are an integral part of the semantics of databases. Currently, the above concepts are being extended to include behaviour using another programming language concept, namely, abstract data types.

Acknowledgements

The author is grateful to J. Schmidt and D. Tsichritzis for their comments on an earlier version of this paper.

References

Abrial, J.R. [1974] Data Semantics. Database Management, Klimbie, J.W. and Koffeman, K.L. (Eds.), North-Holland Publ. Co., Amsterdam, The Netherlands.

ANSI/SPARC [1977] The ANSI/X3/SPARC DBMS Framework. IFIPS Press, Montvale, N.J., U.S.A.

Atkinson, M.P. [1978] Programming languages and data bases. Proc. Int'l Conf. on Very Large Databases, Sept. 1978.

Bachman, C.W. and Daya, M. [1977] The role concept in data models. Proc. Int'l Conf. on Very Large Databases, Oct. 1977.

Bernstein, P.A. [1976] Synthesizing third normal form relations from functional dependencies. ACM TODS 1,4 (Dec. 1976).

Biller, H. and Neuhold, E.J. [1978] Semantics of data bases: The semantics of data models. Information Systems 3, 1 1978.

Brodie, M.L. [1978] Specification and verification of database semantic integrity. Ph.D. thesis, CSRG-91, U. of Toronto.

Brodie, M.L. [1979] Data quality: data reliability and semantic integrity. Proc. INFOTECH Conf. on Data Design, London, September 1979.

Brodie, M.L. and Schmidt, J.W. [1978] What is the use of abstract data types in databases?. Proc. Int'l. Conf. on Very Large Data Bases, Berlin, September 1978.

Codd, E.F. [1970] A relational model for large shared data banks. Comm. ACM 13, 6 (June 1970).

Gotlieb, C.C. and Gotlieb, L.R. [1978] Data Types and Structures, Prentice Hall, Englewood Cliffs, N.J.

Guttag, J.V. [1975] The specification and application to programming of abstract data types. Ph.D. thesis, CSRG-59, U. of Toronto, Sept. 1975.

Hammer, M.M. [1976] Data abstraction for data bases. Proc. Conf. on Data: Abstraction, Definition and Structure, SIGMOD FDT 8, 12 (March 1976).

Hammer, M.M. and McLeod, D.J. [1978] The semantic data model: A modelling mechanism for database applications. Proc. ACM SIGMOD 1978, ACM New York.

Hoare, C.A.R. [1972] Notes on data structuring. APIC Studies in Data Processing No, 8: Structured Programming, Academic Press, New York.

Hoare, C.A.R. [1975] Data reliability. Proc. Int'l Conf. on Reliable Software, SIGPLAN Notices 10, 6 (June 1975).

Hoare, C.A.R. and Wirth, N. [1973] An axiomatic definition of the programming language Pascal. Acta Informatica 2.

Jensen, K. and Wirth, N. [1974] PASCAL User Manual and Report, 2nd Ed., Lecture Notes in Computer Science, 18, Springer-Verlag.

Kent, W. [1979] Limitations of record-oriented information models. ACM TODS 4, 1 (March 1979).

Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G. and Popek, G.J. [1977] Report on the programming language Euclid. SIGPLAN Notices 12, 2 (Feb. 1977).

Liskov, B.H. and Zilles, S.N. [1974] Programming with abstract data types. Proc. Symposium on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974).

Liskov, B.H. and Zilles, S.N. [1975] Specification techniques for data abstractions. Proc. Int'l. Conf. on Reliable Software. SIGPLAN Notices 10, 6 (June 1975).

London, R.L., Guttag, J.V., Horning, J.J., Lampson, B.W., Mitchell, J.G. and Popek, G.J. [1978] Proof rules for the programming language Euclid. to appear Acta Informatica.

McGee, W.C. [1976] On user criteria for data model evaluation. ACM TODS 1, 4 (Dec. 1976).

Morris, J.H. [1973] Types are not sets. Proc. ACM Symposium on Principles of Programmming Languages, Oct. 1973.

Parnas, D.L., Shore, J.E. and Weiss, D. [1976] Abstract data types defined as classes of variables. Proc. Conf. on Data; Abstraction, Definition, and Structure, SIGMOD FDT 8, 12 (March 1976).

Pelagatti, G., Paolinii, P. and Bracchi, G. [1978] Mapping external views to a common data model. Information Systems 3, 2 1978.

Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G. and London, R.L. [1977] Notes on the design of Euclid. SIGPLAN Notices 12, 3 (March 1977).

Roussopoulos, N. [1976] A semantic network model for data bases. Ph.D. thesis, Dept. of Computer Science, U. of Toronto.

Schmid, H.A. and Swenson, R.J. [1975] On the semantics of relational data model. Proc. ACM SIGMOD 1975, ACM New York.

Schmidt, J.W. [1977] Some high level language constructs for data of type relation. ACM TODS 2, 3 (Sept. 1977).

Schmidt, J.W. [1978] Type concepts for database definition. Proc. Int'l Conf. on Databases: Improving Reliability and Responsiveness, Haifa, Israel, (Aug. 1978).

Smith, J.M. and Smith, D.C.P. [1977] Database abstraction: Aggregation and generalization. ACM TODS 2, 2 (June 1977).

Wegbreit, B. [1974] The treatment of data types in EL1. Comm. ACM 17, 5 (May 1974).

Wong, H.K.T. and Mylopoulos, J. [1977] Two views of data semantics: A survey of data models in artificial intelligence and data management. INFOR 15, 3 (Oct. 1977).